**HttpClient**

Most front-end applications communicate with backend services over the HTTP protocol. Modern browsers support two different APIs for making HTTP requests: the XMLHttpRequest interface and the fetch() API.

The HttpClient in @angular/common/http offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequestinterface exposed by browsers. Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

## Setup

Before you can use the HttpClient, you need to import the Angular HttpClientModule. Most apps do so in the root AppModule.

app/app.module.ts (excerpt)

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { HttpClientModule } from '@angular/common/http';

@NgModule({

imports: [

BrowserModule,

// import HttpClientModule after BrowserModule.

HttpClientModule,

],

declarations: [

AppComponent,

],

bootstrap: [ AppComponent ]

})

export class AppModule {}

Having imported HttpClientModule into the AppModule, you can inject the HttpClient into an application class as shown in the following ConfigService example.

app/config/config.service.ts (excerpt)

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

@Injectable()

export class ConfigService {

constructor(private http: HttpClient) { }

}

## Getting JSON data

Applications often request JSON data from the server. For example, the app might need a configuration file on the server, config.json, that specifies resource URLs.

assets/config.json

{

"employeesUrl": "api/employees",

"textfile": "assets/textfile.txt"

}

The ConfigService fetches this file with a get() method on HttpClient.

app/config/config.service.ts (getConfig v.1)

configUrl = 'assets/config.json';

getConfig() {

return this.http.get(this.configUrl);

}

A component, such as ConfigComponent, injects the ConfigService and calls the getConfig service method.

app/config/config.component.ts (showConfig v.1)

showConfig() {

this.configService.getConfig()

.subscribe((data: Config) => this.config = {

employeesUrl: data['employeesUrl'],

textfile: data['textfile']

});

}

Because the service method returns an Observable of configuration data, the component **subscribes** to the method's return value. The subscription callback copies the data fields into the component's config object, which is data-bound in the component template for display.

### Why write a service

This example is so simple that it is tempting to write the Http.get() inside the component itself and skip the service.

However, data access rarely stays this simple. You typically post-process the data, add error handling, and maybe some retry logic to cope with intermittent connectivity.

The component quickly becomes cluttered with data access minutia. The component becomes harder to understand, harder to test, and the data access logic can't be re-used or standardized.

That's why it is a best practice to separate presentation of data from data access by encapsulating data access in a separate service and delegating to that service in the component, even in simple cases like this one.

### Type-checking the response

The subscribe callback above requires bracket notation to extract the data values.

.subscribe((data: Config) => this.config = {

employeesUrl: data['employeesUrl'],

textfile: data['textfile']

});

You can't write data.employeesUrl because TypeScript correctly complains that the data object from the service does not have a employeesUrlproperty.

The HttpClient.get() method parsed the JSON server response into the anonymous Object type. It doesn't know what the shape of that object is.

You can tell HttpClient the type of the response to make consuming the output easier and more obvious.

First, define an interface with the correct shape:

export interface Config {

employeesUrl: string;

textfile: string;

}

Then, specify that interface as the HttpClient.get() call's type parameter in the service:

app/config/config.service.ts (getConfig v.2)

getConfig() {

// now returns an Observable of Config

return this.http.get<Config>(this.configUrl);

}

The callback in the updated component method receives a typed data object, which is easier and safer to consume:

app/config/config.component.ts (showConfig v.2)

config: Config;

showConfig() {

this.configService.getConfig()

// clone the data object, using its known Config shape

.subscribe((data: Config) => this.config = { ...data });

}

### Reading the full response

The response body doesn't return all the data you may need. Sometimes servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

Tell HttpClient that you want the full response with the observe option:

getConfigResponse(): Observable<HttpResponse<Config>> {

return this.http.get<Config>(

this.configUrl, { observe: 'response' });

}

Now HttpClient.get() returns an Observable of typed HttpResponse rather than just the JSON data.

The component's showConfigResponse() method displays the response headers as well as the configuration:

app/config/config.component.ts (showConfigResponse)

showConfigResponse() {

this.configService.getConfigResponse()

// resp is of type `HttpResponse<Config>`

.subscribe(resp => {

// display its headers

const keys = resp.headers.keys();

this.headers = keys.map(key =>

`${key}: ${resp.headers.get(key)}`);

// access the body directly, which is typed as `Config`.

this.config = { ... resp.body };

});

}

As you can see, the response object has a body property of the correct type.

## Error handling

What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server? HttpClient will return an error object instead of a successful response.

You could handle in the component by adding a second callback to the .subscribe():

app/config/config.component.ts (showConfig v.3 with error handling)

showConfig() {

this.configService.getConfig()

.subscribe(

(data: Config) => this.config = { ...data }, // success path

error => this.error = error // error path

);

}

It's certainly a good idea to give the user some kind of feedback when data access fails. But displaying the raw error object returned by HttpClient is far from the best way to do it.

### Getting error details

Detecting that an error occurred is one thing. Interpreting that error and composing a user-friendly response is a bit more involved.

Two types of errors can occur. The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.

Or something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript ErrorEvent objects.

The HttpClient captures both kinds of errors in its HttpErrorResponse and you can inspect that response to figure out what really happened.

Error inspection, interpretation, and resolution is something you want to do in the service, not in the component.

You might first devise an error handler like this one:

app/config/config.service.ts (handleError)

private handleError(error: HttpErrorResponse) {

if (error.error instanceof ErrorEvent) {

// A client-side or network error occurred. Handle it accordingly.

console.error('An error occurred:', error.error.message);

} else {

// The backend returned an unsuccessful response code.

// The response body may contain clues as to what went wrong,

console.error(

`Backend returned code ${error.status}, ` +

`body was: ${error.error}`);

}

// return an observable with a user-facing error message

return throwError(

'Something bad happened; please try again later.');

};

Notice that this handler returns an RxJS ErrorObservable with a user-friendly error message. Consumers of the service expect service methods to return an Observable of some kind, even a "bad" one.

Now you take the Observables returned by the HttpClient methods and pipe them through to the error handler.

app/config/config.service.ts (getConfig v.3 with error handler)

getConfig() {

return this.http.get<Config>(this.configUrl).pipe( catchError(this.handleError) );

}

### retry()

Sometimes the error is transient and will go away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again may produce a successful result.

The RxJS library offers several retry operators that are worth exploring. The simplest is called retry() and it automatically re-subscribes to a failed Observable a specified number of times. Re-subscribing to the result of an HttpClient method call has the effect of reissuing the HTTP request.

Pipe it onto the HttpClient method result just before the error handler.

app/config/config.service.ts (getConfig with retry)

getConfig() {

return this.http.get<Config>(this.configUrl)

.pipe(

retry(3), // retry a failed request up to 3 times

catchError(this.handleError) // then handle the error

);

}

## Observables and operators

The previous sections of this guide referred to RxJS Observables and operators such as catchError and retry. You will encounter more RxJS artifacts as you continue below.

RxJS is a library for composing asynchronous and callback-based code in a functional, reactive style. Many Angular APIs, including HttpClient, produce and consume RxJS Observables.

RxJS itself is out-of-scope for this guide. You will find many learning resources on the web. While you can get by with a minimum of RxJS knowledge, you'll want to grow your RxJS skills over time in order to use HttpClient effectively.

If you're following along with these code snippets, note that you must import the RxJS observable and operator symbols that appear in those snippets. These ConfigService imports are typical.

app/config/config.service.ts (RxJS imports)

import { Observable, throwError } from 'rxjs';

import { catchError, retry } from 'rxjs/operators';

## Requesting non-JSON data

Not all APIs return JSON data. In this next example, a DownloaderService method reads a text file from the server and logs the file contents, before returning those contents to the caller as an Observable<string>.

app/downloader/downloader.service.ts (getTextFile)

getTextFile(filename: string) {

// The Observable returned by get() is of type Observable<string>

// because a text response was specified.

// There's no need to pass a <string> type parameter to get().

return this.http.get(filename, {responseType: 'text'})

.pipe(

tap( // Log the result or error

data => this.log(filename, data),

error => this.logError(filename, error)

)

);

}

HttpClient.get() returns a string rather than the default JSON because of the responseType option.

The RxJS tap operator (as in "wiretap") lets the code inspect good and error values passing through the observable without disturbing them.

A download() method in the DownloaderComponent initiates the request by subscribing to the service method.

app/downloader/downloader.component.ts (download)

download() {

this.downloaderService.getTextFile('assets/textfile.txt')

.subscribe(results => this.contents = results);

}

## Sending data to the server

In addition to fetching data from the server, HttpClient supports mutating requests, that is, sending data to the server with other HTTP methods such as PUT, POST, and DELETE.

The sample app for this guide includes a simplified version of the "Tour of Employees" example that fetches employees and enables users to add, delete, and update them.

The following sections excerpt methods of the sample's EmployeesService.

### Adding headers

Many servers require extra headers for save operations. For example, they may require a "Content-Type" header to explicitly declare the MIME type of the request body. Or perhaps the server requires an authorization token.

The EmployeesService defines such headers in an httpOptions object that will be passed to every HttpClient save method.

app/employees/employees.service.ts (httpOptions)

import { HttpHeaders } from '@angular/common/http';

const httpOptions = {

headers: new HttpHeaders({

'Content-Type': 'application/json',

'Authorization': 'my-auth-token'

})

};

### Making a POST request

Apps often POST data to a server. They POST when submitting a form. In the following example, the EmployeesService posts when adding a Employee to the database.

app/employees/employees.service.ts (addEmployee)

/\*\* POST: add a new Employee to the database \*/

addEmployee (Employee: Employee): Observable<Employee> {

return this.http.post<Employee>(this.employeesUrl, Employee, httpOptions)

.pipe(

catchError(this.handleError('addEmployee', Employee))

);

}

The HttpClient.post() method is similar to get() in that it has a type parameter (you're expecting the server to return the new Employee) and it takes a resource URL.

It takes two more parameters:

1. Employee - the data to POST in the body of the request.
2. httpOptions - the method options which, in this case, specify required headers.

Of course it catches errors in much the same manner described above.

The EmployeesComponent initiates the actual POST operation by subscribing to the Observable returned by this service method.

app/employees/employees.component.ts (addEmployee)

this.employeesService.addEmployee(newEmployee)

.subscribe(Employee => this.employees.push(Employee));

When the server responds successfully with the newly added Employee, the component adds that Employee to the displayed employees list.

### Making a DELETE request

This application deletes a Employee with the HttpClient.delete method by passing the Employee's id in the request URL.

app/employees/employees.service.ts (deleteEmployee)

/\*\* DELETE: delete the Employee from the server \*/

deleteEmployee (id: number): Observable<{}> {

const url = `${this.employeesUrl}/${id}`; // DELETE api/employees/42

return this.http.delete(url, httpOptions)

.pipe(

catchError(this.handleError('deleteEmployee'))

);

}

The EmployeesComponent initiates the actual DELETE operation by subscribing to the Observable returned by this service method.

app/employees/employees.component.ts (deleteEmployee)

this.employeesService.deleteEmployee(Employee.id).subscribe();

The component isn't expecting a result from the delete operation, so it subscribes without a callback. Even though you are not using the result, you still have to subscribe. Calling the subscribe() method executes the observable, which is what initiates the DELETE request.

You must call subscribe() or nothing happens. Just calling EmployeesService.deleteEmployee() **does not initiate the DELETE request.**

// oops ... subscribe() is missing so nothing happens

this.employeesService.deleteEmployee(Employee.id);

**Always**subscribe**!**

An HttpClient method does not begin its HTTP request until you call subscribe() on the observable returned by that method. This is true for all HttpClient methods.

The AsyncPipe subscribes (and unsubscribes) for you automatically.

All observables returned from HttpClient methods are cold by design. Execution of the HTTP request is deferred, allowing you to extend the observable with additional operations such as tap and catchError before anything actually happens.

Calling subscribe(...) triggers execution of the observable and causes HttpClient to compose and send the HTTP request to the server.

You can think of these observables as blueprints for actual HTTP requests.

In fact, each subscribe() initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests.

const req = http.get<Employees>('/api/employees');

// 0 requests made - .subscribe() not called.

req.subscribe();

// 1 request made.

req.subscribe();

// 2 requests made.

### Making a PUT request

An app will send a PUT request to completely replace a resource with updated data. The following EmployeesService example is just like the POST example.

app/employees/employees.service.ts (updateEmployee)

/\*\* PUT: update the Employee on the server. Returns the updated Employee upon success. \*/

updateEmployee (Employee: Employee): Observable<Employee> {

return this.http.put<Employee>(this.employeesUrl, Employee, httpOptions)

.pipe(

catchError(this.handleError('updateEmployee', Employee))

);

}

For the reasons explained above, the caller (EmployeesComponent.update() in this case) must subscribe() to the observable returned from the HttpClient.put() in order to initiate the request.

## Advanced usage

We have discussed the basic HTTP functionality in @angular/common/http, but sometimes you need to do more than make simple requests and get data back.

### Configuring the request

Other aspects of an outgoing request can be configured via the options object passed as the last argument to the HttpClient method.

You saw earlier that the EmployeesService sets the default headers by passing an options object (httpOptions) to its save methods. You can do more.

#### Update headers

You can't directly modify the existing headers within the previous options object because instances of the HttpHeaders class are immutable.

Use the set() method instead. It returns a clone of the current instance with the new changes applied.

Here's how you might update the authorization header (after the old token expired) before making the next request.

httpOptions.headers =

httpOptions.headers.set('Authorization', 'my-new-auth-token');

#### URL Parameters

Adding URL search parameters works a similar way. Here is a searchEmployees method that queries for employees whose names contain the search term.

/\* GET employees whose name contains search term \*/

searchEmployees(term: string): Observable<Employee[]> {

term = term.trim();

// Add safe, URL encoded search parameter if there is a search term

const options = term ?

{ params: new HttpParams().set('name', term) } : {};

return this.http.get<Employee[]>(this.employeesUrl, options)

.pipe(

catchError(this.handleError<Employee[]>('searchEmployees', []))

);

}

If there is a search term, the code constructs an options object with an HTML URL-encoded search parameter. If the term were "foo", the GET request URL would be api/employees/?name=foo.

The HttpParams are immutable so you'll have to use the set() method to update the options.

### Debouncing requests

The sample includes an npm package search feature.

When the user enters a name in a search-box, the PackageSearchComponent sends a search request for a package with that name to the NPM web API.

Here's a pertinent excerpt from the template:

app/package-search/package-search.component.html (search)

<input (keyup)="search($event.target.value)" id="name" placeholder="Search"/>

<ul>

<li \*ngFor="let package of packages$ | async">

<b>{{package.name}} v.{{package.version}}</b> -

<i>{{package.description}}</i>

</li>

</ul>

The (keyup) event binding sends every keystroke to the component's search() method.

Sending a request for every keystroke could be expensive. It's better to wait until the user stops typing and then send a request. That's easy to implement with RxJS operators, as shown in this excerpt.

app/package-search/package-search.component.ts (excerpt))

1. withRefresh = false;
2. packages$: Observable<NpmPackageInfo[]>;
3. private searchText$ = new Subject<string>();
5. search(packageName: string) {
6. this.searchText$.next(packageName);
7. }
9. ngOnInit() {
10. this.packages$ = this.searchText$.pipe(
11. debounceTime(500),
12. distinctUntilChanged(),
13. switchMap(packageName =>
14. this.searchService.search(packageName, this.withRefresh))
15. );
16. }
18. constructor(private searchService: PackageSearchService) { }

The searchText$ is the sequence of search-box values coming from the user. It's defined as an RxJS Subject, which means it is a multicasting Observable that can also produce values for itself by calling next(value), as happens in the search() method.

Rather than forward every searchText value directly to the injected PackageSearchService, the code in ngOnInit() pipes search values through three operators:

1. debounceTime(500) - wait for the user to stop typing (1/2 second in this case).
2. distinctUntilChanged() - wait until the search text changes.
3. switchMap() - send the search request to the service.

The code sets packages$ to this re-composed Observable of search results. The template subscribes to packages$ with the AsyncPipeand displays search results as they arrive.

A search value reaches the service only if it's a new value and the user has stopped typing.

The withRefresh option is explained below.

#### switchMap()

The switchMap() operator has three important characteristics.

1. It takes a function argument that returns an Observable. PackageSearchService.search returns an Observable, as other data service methods do.
2. If a previous search request is still in-flight (as when the connection is poor), it cancels that request and sends a new one.
3. It returns service responses in their original request order, even if the server returns them out of order.

If you think you'll reuse this debouncing logic, consider moving it to a utility function or into the PackageSearchService itself.

### Intercepting requests and responses

HTTP Interception is a major feature of @angular/common/http. With interception, you declare interceptors that inspect and transform HTTP requests from your application to the server. The same interceptors may also inspect and transform the server's responses on their way back to the application. Multiple interceptors form a forward-and-backward chain of request/response handlers.

Interceptors can perform a variety of implicit tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response.

Without interception, developers would have to implement these tasks explicitly for each HttpClient method call.

#### Write an interceptor

To implement an interceptor, declare a class that implements the intercept() method of the HttpInterceptor interface.

Here is a do-nothing noop interceptor that simply passes the request through without touching it:app/http-interceptors/noop-interceptor.ts

import { Injectable } from '@angular/core';

import {

HttpEvent, HttpInterceptor, HttpHandler, HttpRequest

} from '@angular/common/http';

import { Observable } from 'rxjs';

/\*\* Pass untouched request through to the next request handler. \*/

@Injectable()

export class NoopInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler):

Observable<HttpEvent<any>> {

return next.handle(req);

}

}

The intercept method transforms a request into an Observable that eventually returns the HTTP response. In this sense, each interceptor is fully capable of handling the request entirely by itself.

Most interceptors inspect the request on the way in and forward the (perhaps altered) request to the handle() method of the next object which implements the HttpHandler interface.

export abstract class HttpHandler {

abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;

}

Like intercept(), the handle() method transforms an HTTP request into an Observable of HttpEvents which ultimately include the server's response. The intercept() method could inspect that observable and alter it before returning it to the caller.

This no-op interceptor simply calls next.handle() with the original request and returns the observable without doing a thing.

#### The next object

The next object represents the next interceptor in the chain of interceptors. The final next in the chain is the HttpClient backend handler that sends the request to the server and receives the server's response.

Most interceptors call next.handle() so that the request flows through to the next interceptor and, eventually, the backend handler. An interceptor could skip calling next.handle(), short-circuit the chain, and return its own Observable with an artificial server response.

This is a common middleware pattern found in frameworks such as Express.js.

#### Provide the interceptor

The NoopInterceptor is a service managed by Angular's dependency injection (DI) system. Like other services, you must provide the interceptor class before the app can use it.

Because interceptors are (optional) dependencies of the HttpClient service, you must provide them in the same injector (or a parent of the injector) that provides HttpClient. Interceptors provided after DI creates the HttpClient are ignored.

This app provides HttpClient in the app's root injector, as a side-effect of importing the HttpClientModule in AppModule. You should provide interceptors in AppModule as well.

After importing the HTTP\_INTERCEPTORS injection token from @angular/common/http, write the NoopInterceptor provider like this:

{ provide: HTTP\_INTERCEPTORS, useClass: NoopInterceptor, multi: true },

Note the multi: true option. This required setting tells Angular that HTTP\_INTERCEPTORS is a token for a multiprovider that injects an array of values, rather than a single value.

You could add this provider directly to the providers array of the AppModule. However, it's rather verbose and there's a good chance that you'll create more interceptors and provide them in the same way. You must also pay close attention to the order in which you provide these interceptors.

Consider creating a "barrel" file that gathers all the interceptor providers into an httpInterceptorProviders array, starting with this first one, the NoopInterceptor.

app/http-interceptors/index.ts

/\* "Barrel" of Http Interceptors \*/

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

import { NoopInterceptor } from './noop-interceptor';

/\*\* Http interceptor providers in outside-in order \*/

export const httpInterceptorProviders = [

{ provide: HTTP\_INTERCEPTORS, useClass: NoopInterceptor, multi: true },

];

Then import and add it to the AppModule providers array like this:

app/app.module.ts (interceptor providers)

providers: [

httpInterceptorProviders

],

As you create new interceptors, add them to the httpInterceptorProviders array and you won't have to revisit the AppModule.

There are many more interceptors in the complete sample code.

#### Interceptor order

Angular applies interceptors in the order that you provide them. If you provide interceptors A, then B, then C, requests will flow in A->B->Cand responses will flow out C->B->A.

You cannot change the order or remove interceptors later. If you need to enable and disable an interceptor dynamically, you'll have to build that capability into the interceptor itself.

#### HttpEvents

You may have expected the intercept() and handle() methods to return observables of HttpResponse<any> as most HttpClientmethods do.

Instead they return observables of HttpEvent<any>.

That's because interceptors work at a lower level than those HttpClient methods. A single HTTP request can generate multiple events, including upload and download progress events. The HttpResponse class itself is actually an event, whose type is HttpEventType.HttpResponseEvent.

Many interceptors are only concerned with the outgoing request and simply return the event stream from next.handle() without modifying it.

But interceptors that examine and modify the response from next.handle() will see all of these events. Your interceptor should return every event untouched unless it has a compelling reason to do otherwise.

#### Immutability

Although interceptors are capable of mutating requests and responses, the HttpRequest and HttpResponse instance properties are readonly, rendering them largely immutable.

They are immutable for a good reason: the app may retry a request several times before it succeeds, which means that the interceptor chain may re-process the same request multiple times. If an interceptor could modify the original request object, the re-tried operation would start from the modified request rather than the original. Immutability ensures that interceptors see the same request for each try.

TypeScript will prevent you from setting HttpRequest readonly properties.

// Typescript disallows the following assignment because req.url is readonly

req.url = req.url.replace('http://', 'https://');

To alter the request, clone it first and modify the clone before passing it to next.handle(). You can clone and modify the request in a single step as in this example.

app/http-interceptors/ensure-https-interceptor.ts (excerpt)

// clone request and replace 'http://' with 'https://' at the same time

const secureReq = req.clone({

url: req.url.replace('http://', 'https://')

});

// send the cloned, "secure" request to the next handler.

return next.handle(secureReq);

The clone() method's hash argument allows you to mutate specific properties of the request while copying the others.

##### The request body

The readonly assignment guard can't prevent deep updates and, in particular, it can't prevent you from modifying a property of a request body object.

req.body.name = req.body.name.trim(); // bad idea!

If you must mutate the request body, copy it first, change the copy, clone() the request, and set the clone's body with the new body, as in the following example.

app/http-interceptors/trim-name-interceptor.ts (excerpt)

// copy the body and trim whitespace from the name property

const newBody = { ...body, name: body.name.trim() };

// clone request and set its body

const newReq = req.clone({ body: newBody });

// send the cloned request to the next handler.

return next.handle(newReq);

##### Clearing the request body

Sometimes you need to clear the request body rather than replace it. If you set the cloned request body to undefined, Angular assumes you intend to leave the body as is. That is not what you want. If you set the cloned request body to null, Angular knows you intend to clear the request body.

newReq = req.clone({ ... }); // body not mentioned => preserve original body

newReq = req.clone({ body: undefined }); // preserve original body

newReq = req.clone({ body: null }); // clear the body

#### Set default headers

Apps often use an interceptor to set default headers on outgoing requests.

The sample app has an AuthService that produces an authorization token. Here is its AuthInterceptor that injects that service to get the token and adds an authorization header with that token to every outgoing request:

app/http-interceptors/auth-interceptor.ts

1. import { AuthService } from '../auth.service';
3. @Injectable()
4. export class AuthInterceptor implements HttpInterceptor {
6. constructor(private auth: AuthService) {}
8. intercept(req: HttpRequest<any>, next: HttpHandler) {
9. // Get the auth token from the service.
10. const authToken = this.auth.getAuthorizationToken();
12. // Clone the request and replace the original headers with
13. // cloned headers, updated with the authorization.
14. const authReq = req.clone({
15. headers: req.headers.set('Authorization', authToken)
16. });
18. // send cloned request with header to the next handler.
19. return next.handle(authReq);
20. }
21. }

The practice of cloning a request to set new headers is so common that there's a setHeaders shortcut for it:

// Clone the request and set the new header in one step.

const authReq = req.clone({ setHeaders: { Authorization: authToken } });

An interceptor that alters headers can be used for a number of different operations, including:

* Authentication/authorization
* Caching behavior; for example, If-Modified-Since
* XSRF protection

#### Logging

Because interceptors can process the request and response together, they can do things like time and log an entire HTTP operation.

Consider the following LoggingInterceptor, which captures the time of the request, the time of the response, and logs the outcome with the elapsed time with the injected MessageService.

app/http-interceptors/logging-interceptor.ts)

1. import { finalize, tap } from 'rxjs/operators';
2. import { MessageService } from '../message.service';
4. @Injectable()
5. export class LoggingInterceptor implements HttpInterceptor {
6. constructor(private messenger: MessageService) {}
8. intercept(req: HttpRequest<any>, next: HttpHandler) {
9. const started = Date.now();
10. let ok: string;
12. // extend server response observable with logging
13. return next.handle(req)
14. .pipe(
15. tap(
16. // Succeeds when there is a response; ignore other events
17. event => ok = event instanceof HttpResponse ? 'succeeded' : '',
18. // Operation failed; error is an HttpErrorResponse
19. error => ok = 'failed'
20. ),
21. // Log when response observable either completes or errors
22. finalize(() => {
23. const elapsed = Date.now() - started;
24. const msg = `${req.method} "${req.urlWithParams}"
25. ${ok} in ${elapsed} ms.`;
26. this.messenger.add(msg);
27. })
28. );
29. }
30. }

The RxJS tap operator captures whether the request succeed or failed. The RxJS finalize operator is called when the response observable either errors or completes (which it must), and reports the outcome to the MessageService.

Neither tap nor finalize touch the values of the observable stream returned to the caller.

#### Caching

Interceptors can handle requests by themselves, without forwarding to next.handle().

For example, you might decide to cache certain requests and responses to improve performance. You can delegate caching to an interceptor without disturbing your existing data services.

The CachingInterceptor demonstrates this approach.

app/http-interceptors/caching-interceptor.ts)

@Injectable()

export class CachingInterceptor implements HttpInterceptor {

constructor(private cache: RequestCache) {}

intercept(req: HttpRequest<any>, next: HttpHandler) {

// continue if not cachable.

if (!isCachable(req)) { return next.handle(req); }

const cachedResponse = this.cache.get(req);

return cachedResponse ?

of(cachedResponse) : sendRequest(req, next, this.cache);

}

}

The isCachable() function determines if the request is cachable. In this sample, only GET requests to the npm package search api are cachable.

If the request is not cachable, the interceptor simply forwards the request to the next handler in the chain.

If a cachable request is found in the cache, the interceptor returns an of() observable with the cached response, by-passing the nexthandler (and all other interceptors downstream).

If a cachable request is not in cache, the code calls sendRequest.

1. /\*\*
2. \* Get server response observable by sending request to `next()`.
3. \* Will add the response to the cache on the way out.
4. \*/
5. function sendRequest(
6. req: HttpRequest<any>,
7. next: HttpHandler,
8. cache: RequestCache): Observable<HttpEvent<any>> {
10. // No headers allowed in npm search request
11. const noHeaderReq = req.clone({ headers: new HttpHeaders() });
13. return next.handle(noHeaderReq).pipe(
14. tap(event => {
15. // There may be other events besides the response.
16. if (event instanceof HttpResponse) {
17. cache.put(req, event); // Update the cache.
18. }
19. })
20. );
21. }

The sendRequest function creates a request clone without headers because the npm api forbids them.

It forwards that request to next.handle() which ultimately calls the server and returns the server's response.

Note how sendRequest intercepts the response on its way back to the application. It pipes the response through the tap() operator, whose callback adds the response to the cache.

The original response continues untouched back up through the chain of interceptors to the application caller.

Data services, such as PackageSearchService, are unaware that some of their HttpClient requests actually return cached responses.

#### Return a multi-valued Observable

The HttpClient.get() method normally returns an observable that either emits the data or an error. Some folks describe it as a "one and done" observable.

But an interceptor can change this to an observable that emits more than once.

A revised version of the CachingInterceptor optionally returns an observable that immediately emits the cached response, sends the request to the NPM web API anyway, and emits again later with the updated search results.

// cache-then-refresh

if (req.headers.get('x-refresh')) {

const results$ = sendRequest(req, next, this.cache);

return cachedResponse ?

results$.pipe( startWith(cachedResponse) ) :

results$;

}

// cache-or-fetch

return cachedResponse ?

of(cachedResponse) : sendRequest(req, next, this.cache);

The cache-then-refresh option is triggered by the presence of a **custom**x-refresh**header**.

A checkbox on the PackageSearchComponent toggles a withRefresh flag, which is one of the arguments to PackageSearchService.search(). That search() method creates the custom x-refresh header and adds it to the request before calling HttpClient.get().

The revised CachingInterceptor sets up a server request whether there's a cached value or not, using the same sendRequest() method described above. The results$ observable will make the request when subscribed.

If there's no cached value, the interceptor returns results$.

If there is a cached value, the code pipes the cached response onto results$, producing a recomposed observable that emits twice, the cached response first (and immediately), followed later by the response from the server. Subscribers see a sequence of two responses.

### Listening to progress events

Sometimes applications transfer large amounts of data and those transfers can take a long time. File uploads are a typical example. Give the users a better experience by providing feedback on the progress of such transfers.

To make a request with progress events enabled, you can create an instance of HttpRequest with the reportProgress option set true to enable tracking of progress events.

app/uploader/uploader.service.ts (upload request)

const req = new HttpRequest('POST', '/upload/file', file, {

reportProgress: true

});

Every progress event triggers change detection, so only turn them on if you truly intend to report progress in the UI.

Next, pass this request object to the HttpClient.request() method, which returns an Observable of HttpEvents, the same events processed by interceptors:

app/uploader/uploader.service.ts (upload body)

// The `HttpClient.request` API produces a raw event stream

// which includes start (sent), progress, and response events.

return this.http.request(req).pipe(

map(event => this.getEventMessage(event, file)),

tap(message => this.showProgress(message)),

last(), // return last (completed) message to caller

catchError(this.handleError(file))

);

The getEventMessage method interprets each type of HttpEvent in the event stream.

app/uploader/uploader.service.ts (getEventMessage)

/\*\* Return distinct message for sent, upload progress, & response events \*/

private getEventMessage(event: HttpEvent<any>, file: File) {

switch (event.type) {

case HttpEventType.Sent:

return `Uploading file "${file.name}" of size ${file.size}.`;

case HttpEventType.UploadProgress:

// Compute and show the % done:

const percentDone = Math.round(100 \* event.loaded / event.total);

return `File "${file.name}" is ${percentDone}% uploaded.`;

case HttpEventType.Response:

return `File "${file.name}" was completely uploaded!`;

default:

return `File "${file.name}" surprising upload event: ${event.type}.`;

}

}

The sample app for this guide doesn't have a server that accepts uploaded files. The UploadInterceptor in app/http-interceptors/upload-interceptor.ts intercepts and short-circuits upload requests by returning an observable of simulated events.

## Security: XSRF Protection

Cross-Site Request Forgery (XSRF) is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. HttpClient supports a common mechanism used to prevent XSRF attacks. When performing HTTP requests, an interceptor reads a token from a cookie, by default XSRF-TOKEN, and sets it as an HTTP header, X-XSRF-TOKEN. Since only code that runs on your domain could read the cookie, the backend can be certain that the HTTP request came from your client application and not an attacker.

By default, an interceptor sends this cookie on all mutating requests (POST, etc.) to relative URLs but not on GET/HEAD requests or on requests with an absolute URL.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called XSRF-TOKEN on either the page load or the first GET request. On subsequent requests the server can verify that the cookie matches the X-XSRF-TOKEN HTTP header, and therefore be sure that only code running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server; this prevents the client from making up its own tokens. Set the token to a digest of your site's authentication cookie with a salt for added security.

In order to prevent collisions in environments where multiple Angular apps share the same domain or subdomain, give each application a unique cookie name.

Note that *HttpClient* supports only the client half of the XSRF protection scheme. Your backend service must be configured to set the cookie for your page, and to verify that the header is present on all eligible requests. If not, Angular's default protection will be ineffective.

### Configuring custom cookie/header names

If your backend service uses different names for the XSRF token cookie or header, use HttpClientXsrfModule.withOptions() to override the defaults.

imports: [

HttpClientModule,

HttpClientXsrfModule.withOptions({

cookieName: 'My-Xsrf-Cookie',

headerName: 'My-Xsrf-Header',

}),

],

## Testing HTTP requests

Like any external dependency, the HTTP backend needs to be mocked so your tests can simulate interaction with a remote server. The @angular/common/http/testing library makes setting up such mocking straightforward.

### Mocking philosophy

Angular's HTTP testing library is designed for a pattern of testing wherein the app executes code and makes requests first.

Then a test expects that certain requests have or have not been made, performs assertions against those requests, and finally provide responses by "flushing" each expected request.

At the end, tests may verify that the app has made no unexpected requests.

You can run these sample tests / download example in a live coding environment.

The tests described in this guide are in src/testing/http-client.spec.ts. There are also tests of an application data service that call HttpClient in src/app/employees/employees.service.spec.ts.

### Setup

To begin testing calls to HttpClient, import the HttpClientTestingModule and the mocking controller, HttpTestingController, along with the other symbols your tests require.

app/testing/http-client.spec.ts (imports)

// Http testing module and mocking controller

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

// Other imports

import { TestBed } from '@angular/core/testing';

import { HttpClient, HttpErrorResponse } from '@angular/common/http';

Then add the HttpClientTestingModule to the TestBed and continue with the setup of the service-under-test.

app/testing/http-client.spec.ts(setup)

describe('HttpClient testing', () => {

let httpClient: HttpClient;

let httpTestingController: HttpTestingController;

beforeEach(() => {

TestBed.configureTestingModule({

imports: [ HttpClientTestingModule ]

});

// Inject the http service and test controller for each test

httpClient = TestBed.get(HttpClient);

httpTestingController = TestBed.get(HttpTestingController);

});

/// Tests begin ///

});

Now requests made in the course of your tests will hit the testing backend instead of the normal backend.

This setup also calls TestBed.get() to inject the HttpClient service and the mocking controller so they can be referenced during the tests.

### Expecting and answering requests

Now you can write a test that expects a GET Request to occur and provides a mock response.

app/testing/http-client.spec.ts(httpClient.get)

it('can test HttpClient.get', () => {

const testData: Data = {name: 'Test Data'};

// Make an HTTP GET request

httpClient.get<Data>(testUrl)

.subscribe(data =>

// When observable resolves, result should match test data

expect(data).toEqual(testData)

);

// The following `expectOne()` will match the request's URL.

// If no requests or multiple requests matched that URL

// `expectOne()` would throw.

const req = httpTestingController.expectOne('/data');

// Assert that the request is a GET.

expect(req.request.method).toEqual('GET');

// Respond with mock data, causing Observable to resolve.

// Subscribe callback asserts that correct data was returned.

req.flush(testData);

// Finally, assert that there are no outstanding requests.

httpTestingController.verify();

});

The last step, verifying that no requests remain outstanding, is common enough for you to move it into an afterEach() step:

afterEach(() => {

// After every test, assert that there are no more pending requests.

httpTestingController.verify();

});

#### Custom request expectations

If matching by URL isn't sufficient, it's possible to implement your own matching function. For example, you could look for an outgoing request that has an authorization header:

// Expect one request with an authorization header

const req = httpTestingController.expectOne(

req => req.headers.has('Authorization')

);

As with the previous expectOne(), the test will fail if 0 or 2+ requests satisfy this predicate.

#### Handling more than one request

If you need to respond to duplicate requests in your test, use the match() API instead of expectOne(). It takes the same arguments but returns an array of matching requests. Once returned, these requests are removed from future matching and you are responsible for flushing and verifying them.

// get all pending requests that match the given URL

const requests = httpTestingController.match(testUrl);

expect(requests.length).toEqual(3);

// Respond to each request with different results

requests[0].flush([]);

requests[1].flush([testData[0]]);

requests[2].flush(testData);

### Testing for errors

You should test the app's defenses against HTTP requests that fail.

Call request.flush() with an error message, as seen in the following example.

it('can test for 404 error', () => {

const emsg = 'deliberate 404 error';

httpClient.get<Data[]>(testUrl).subscribe(

data => fail('should have failed with the 404 error'),

(error: HttpErrorResponse) => {

expect(error.status).toEqual(404, 'status');

expect(error.error).toEqual(emsg, 'message');

}

);

const req = httpTestingController.expectOne(testUrl);

// Respond with mock error

req.flush(emsg, { status: 404, statusText: 'Not Found' });

});

Alternatively, you can call request.error() with an ErrorEvent.

it('can test for network error', () => {

const emsg = 'simulated network error';

httpClient.get<Data[]>(testUrl).subscribe(

data => fail('should have failed with the network error'),

(error: HttpErrorResponse) => {

expect(error.error.message).toEqual(emsg, 'message');

}

);

const req = httpTestingController.expectOne(testUrl);

// Create mock ErrorEvent, raised when something goes wrong at the network level.

// Connection timeout, DNS error, offline, etc

const mockError = new ErrorEvent('Network error', {

message: emsg,

});

// Respond with mock error

req.error(mockError);

});